

IA32 OS START-UP UEFI FIRMWARE

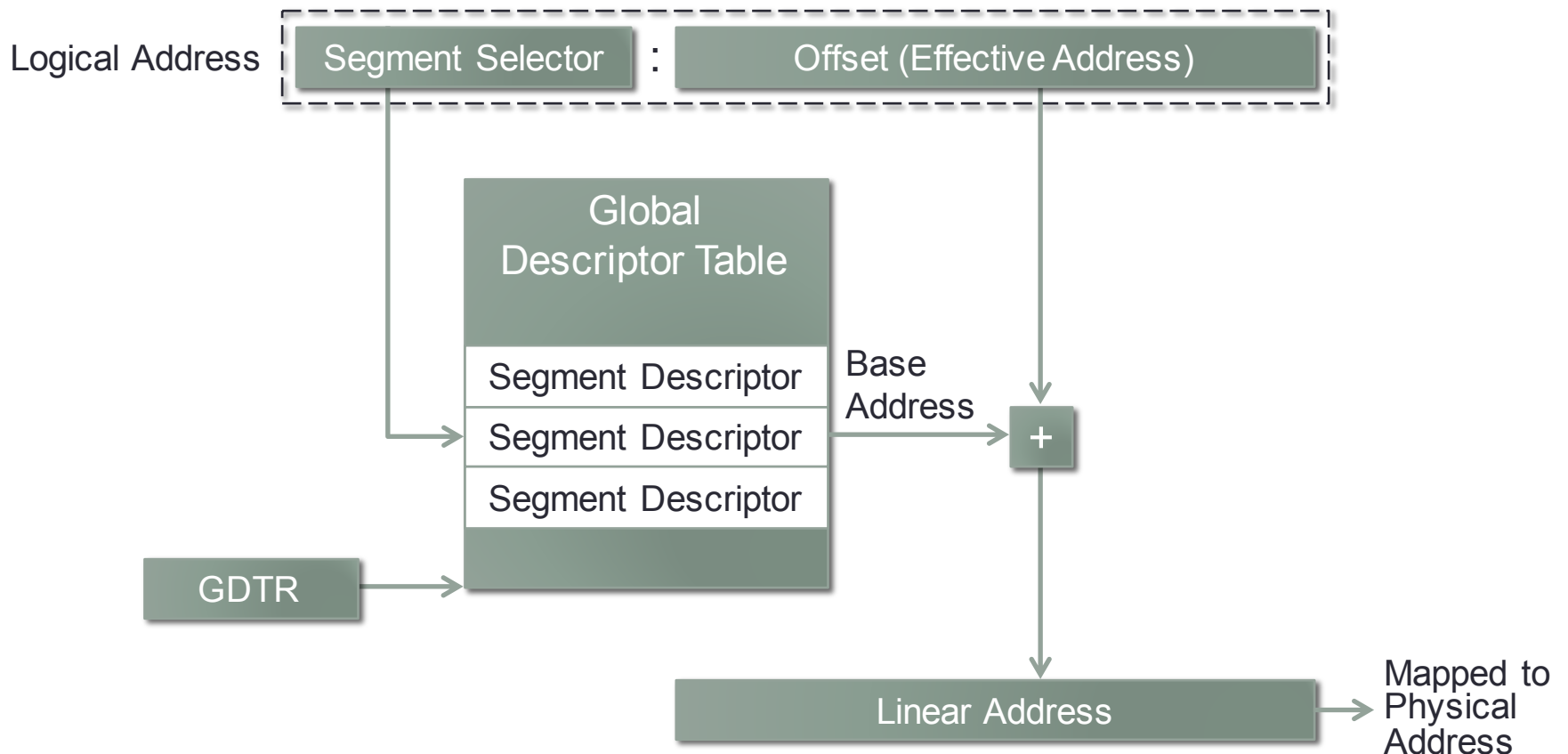
CS124 – Operating Systems
Winter 2015-2016, Lecture 6

Last Time: IA32 Bootstrap

- Computers and operating systems employ a bootstrap process to load and start the operating system
 - A series of increasingly complex boot loaders
 - Each stage is responsible for loading the next stage of the process
- Partially covered the IA32 bootstrap process
 - Details so far were grungy, due to IA32 remaining backward compatible all the way to original 8086/8088 processor
- IA32 has segmented memory, multiple addressing modes
- Real-address mode (aka “real mode”):
 - Supports a 20-bit address space
 - Segment selectors are 16 bits; offsets are 16 bits
 - $address = segment \ll 4 + offset$

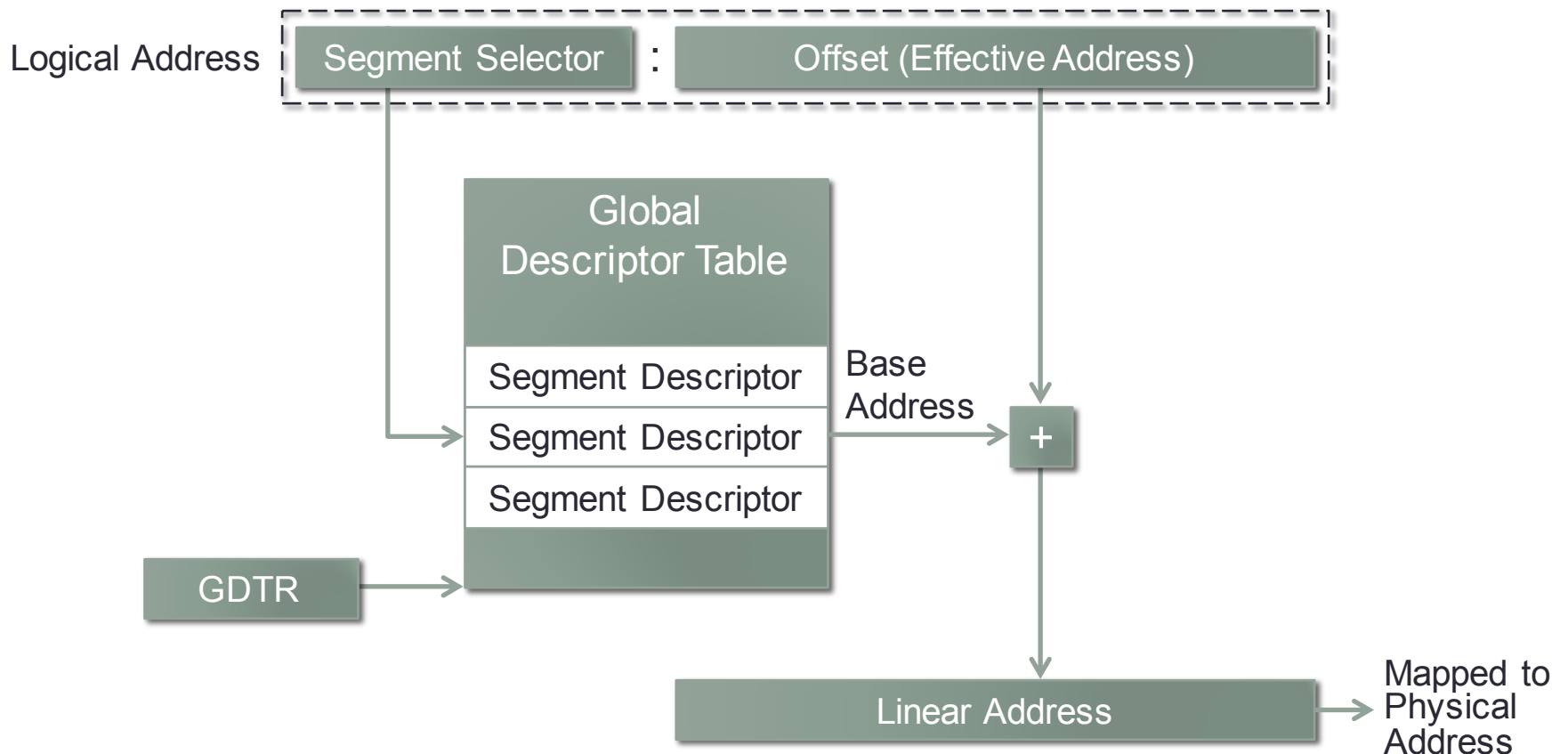
Last Time: IA32 Bootstrap (2)

- In protected mode, segment selectors are indexes into the Global Descriptor Table
 - Specifies start and length of segments, protection levels, etc.



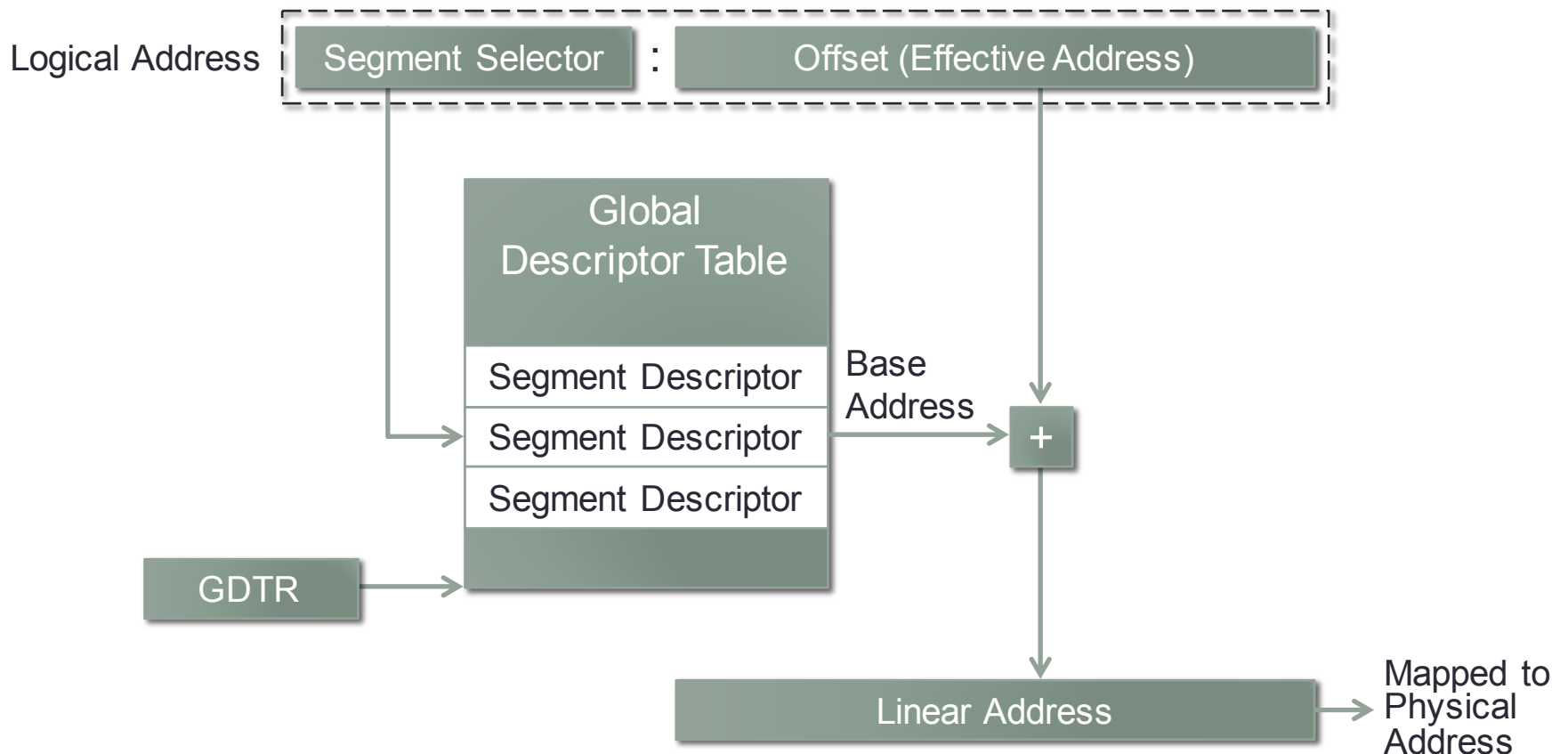
Last Time: IA32 Bootstrap (3)

- Most OSes use a flat memory model
 - All segments start at beginning of linear address space
 - All segments end at the end of linear address space



Last Time: IA32 Bootstrap (4)

- Linear address is mapped to a physical address using the current virtual-memory page table configuration
 - OSes use virtual memory to isolate process' data from each other



IA32 Bootstrap Process So Far...

- At power on, IA32 processor starts executing instructions at the address 0xFFFFFFFF0
 - A ROM memory is positioned at this hardware address to jump to BIOS bootstrap code
- BIOS bootstrap code performs a power-on self test if it was a hardware reset (skips if software reset)
- Next, BIOS attempts to load the first 512-byte sector of each bootable device in the system at address 0x7C00
 - On success, BIOS jumps to address 0x7C00 for next stage of boot
- If it's an MBR bootloader, it must chain-load the boot sector for the operating system to start
- Finally, the OS bootloader must load the OS kernel into memory and jump to the kernel bootstrap code

From Real to Protected Mode

- After the IA32 OS bootloader loads the kernel, it must switch from real-addressing mode to protected mode
 - This is technically “kernel startup code”
- A few other annoyances to deal with (of course)
- Original 8086 had a 1MiB address space, but real-addressing mode allows addresses beyond this range
 - e.g. FFFF:FFFF = 10FFEF (~65500 bytes past the 1MiB barrier)
 - 8086 only had 20 address lines (A0..A19), so addresses beyond this range would simply wrap around
 - Unfortunately, some programs relied on this behavior ☹
- 80286 forward could access more than 1MiB of memory...
 - To remain backward-compatible, the A20 address line is disabled and forced to 0 at startup.
 - (And, they made this configurable by running the A20 line through the keyboard controller chip...)

From Real to Protected Mode (2)

- Modern IA32 computers often still leave the A20 address line disabled at startup
- Step 1: Startup code must re-enable the A20 address line so it can access > 1MiB of memory
 - Some earlier-stage bootloaders take care of this (e.g. GRUB)
 - Some BIOSes also take care of this
 - Startup code has to check if A20 is disabled, and if so, reenabling it
- Step 2: Startup code must configure the protected-mode memory segments, and virtual memory system
 - At the very least, must initialize kernel-data and kernel-code segment descriptors, and set `%cs`, `%ds`, `%es` and `%ss` appropriately
 - May also want to set up a basic virtual memory page-table hierarchy for mapping linear addresses to physical addresses

From Real to Protected Mode (3)

- Step 3: Switch from real mode to protected mode
 - This step is more complicated than you might think...
- Roughly, the process goes like this (a few details omitted):
 1. Disable interrupts! If any interrupts occur during the transition, all mayhem will break loose.
 2. Load the Global Descriptor Table Register (GDTR) with a pointer to the GDT containing the OS' segment descriptors
 3. Load the Task Register (TR) with a simple Task State Segment, so that protected-mode interrupt handling will work properly
 4. Turn on protected mode (and optionally, enable the virtual memory paging system) by writing to control-register `%cr0`
 - (If paging is enabled, must also set up an initial page table via `%cr3`)

From Real to Protected Mode (4)

- Step 3: Switch from real mode to protected mode (cont.)
 - At this point, protected mode is enabled, but the startup code is still running against a 16-bit real-mode segment cached in the CPU
 - 5. Force the CPU to load the new 32-bit protected mode segment selectors by performing a long-jump to the next instruction
 - Long-jump specifies the new kernel-code segment selector value, which also loads this segment selector into `%cs`
 - 6. Set the other segment registers to the kernel-data segment selector value
 - 7. Load the Interrupt Descriptor Table Register (IDTR) with a pointer to the interrupt descriptor table for the operating system
 - 8. Reenable interrupts!
- Once this is done, the OS kernel is ready to take over

Protected Mode and BIOS

- Once the system is in protected mode, we can no longer use BIOS functions to interact with the hardware
 - Problem: BIOS uses (and requires) IA32 real-addressing mode
- From this point forward, the operating system must use its own **device drivers** to interact with computer hardware
 - Software components that know how to interact with a specific kind of device, but that also present a simple, generic interface
- OS device drivers often reinitialize the hardware to suit the needs/preferences of the operating system

Review: IA32 Bootloading

- Steps that are completed to load the operating system:
 1. CPU starts executing instructions at 0xFFFFFFF0
 2. ROM at this address jumps to BIOS bootstrap program
 3. BIOS bootstrap program performs basic initialization, then loads a boot-sector off of the first bootable device that has one
 4. 512-byte boot sector either loads the OS kernel directly, or it loads a 2nd-stage bootloader that loads the kernel
 - e.g. LILO, GRUB, Windows NTLDR, etc.
 5. Kernel startup code completes hardware initialization, then starts loading and running the operating system

Final BIOS Notes

- Windows endeavors to maintain backward compatibility with MS-DOS programs and other older programs
- Windows replaces the BIOS interrupt handlers with its own handlers that call into the Windows APIs
 - DOS programs run under Windows can rely on expected BIOS entry-points
 - Windows can still provide crucial OS services like file permissions, process isolation, etc.

IA32 Bootloading Challenges

- IA32 bootloading is an incredibly baroque process, due to:
 - IA32 backward compatibility, all the way back to 8086
 - BIOS services aren't useable in protected mode
 - Widely varying mechanisms for performing necessary tasks
- Example: enabling A20 address line has three options:
 - A slow and tedious interaction with the keyboard controller
 - A fast mechanism via a System Control Port
 - An extended BIOS call provided in some BIOSes
 - ...that is, if the BIOS or the bootloader hasn't already enabled it...
- Other hardware devices can be equally difficult
 - Many different ways to detect and configure devices in the system
 - Limitations on hardware interrupt lines to be assigned to devices

Plug and Play

- Over time, several standards were published to make PC systems more “**plug-and-play**” capable
 - Allow OS to identify and configure hardware devices automatically, via software
- The hardware bus must support this mechanism:
 - Allow vendors to specify device ID and type values that can be read by the CPU
 - When system buses are initialized, system can enumerate devices connected to the bus and handle each device’s basic initialization
- Example plug-and-play buses:
 - PCI family of buses (PCI, PCI Express, Mini PCI, etc.)
 - USB, FireWire
 - PC Card/PCMCIA (for removable laptop peripherals)

Plug and Play (2)

- With hardware that facilitates device discovery, systems began providing more detailed information to the OS
- Frequently exposed as tables of data set up by the BIOS during bootstrap
- Example: Intel MultiProcessor Specification (1997)
 - Identifies processor manufacturer, model number, etc.
 - Identifies all system buses, processors, processor APIC IDs, etc.
 - Table is set up by the BIOS at startup time
 - A multiprocessor operating system can locate this table and use it to run processes on all available processors

ACPI Standard

- One of the more notable standards is the ACPI standard
 - Advanced Configuration and Power Interface
 - Defines a platform-independent interface for hardware discovery, configuration, power management and monitoring
 - Replaces several previous standards
- ACPI primarily consists of a large number of tables that contain platform configuration details
- All tables are accessible through a structure called the Root System Description Pointer
- Tables include details for all major aspects of the system
- Tables are initialized by bootstrap firmware
 - e.g. multicore/multiprocessor and APIC details
 - e.g. memory characteristics and memory topology

ACPI Tables

- OS must search for ACPI Root System Descriptor Pointer
 - Can be stored in first 1KiB of Extended BIOS Data Area
 - (Address of Extended BIOS area is usually stored at address 0x040E)
 - Or, can be stored somewhere in address range 0xE0000-0xFFFFF (main BIOS area)
- To find the Root System Descriptor Pointer (RSDP):
 - It starts on a 16-byte boundary, with 8-byte signature "RSD PTR "
 - When a candidate for the RSDP is found, compute the structure's checksum to verify. If the checksum is 0, you found it!
- Can follow the RSDP to all other ACPI tables
 - Must compute checksum on each table to verify that it's valid
- Provides a significant amount of discoverability to the OS for determining and controlling hardware configuration

Unified Extensible Firmware Interface

- Data tables are helpful...
- BIOS is still very limiting for modern OSes to deal with
 - Can't even use it after switching to protected mode
- Recently, a new standard has emerged: Unified Extensible Firmware Interface (UEFI)
 - Completely replaces the old BIOS interface with a new, modular, extensible firmware
- Prompted by Intel Itanium processor
 - 64-bit processor, couldn't run BIOS!
 - Still needed to support an operating system
- UEFI is a firmware interface standard
 - Sits on top of lower-level firmware, not directly on computer hardware

Bootloader and OS

UEFI

Firmware

Hardware

Unified Extensible Firmware Interface (2)

- UEFI is a modular system, allowing components to be installed and removed
- Can install UEFI bootloaders for OSes on the computer
 - Knows how to use UEFI services to load and run the OS
- Can install UEFI applications that allow system hardware, boot configuration, etc. to be managed
 - Runs in the “preboot environment” (before the OS is started)
 - e.g. UEFI systems usually have a command shell for basic tasks
 - UEFI bootloaders (a.k.a. OS loaders) are one kind of application
- UEFI drivers provide standardized abstractions for hardware including buses and devices
 - Used by UEFI applications and OS loaders to perform their tasks

Unified Extensible Firmware Interface (3)

- Many services are provided via UEFI protocols
 - For interacting with specific devices, or with specific kinds of data
 - e.g. load images, display images on screen, TCP/IP networking, decompress data, and many more
- Services divided into boot services and runtime services
 - Boot services are only available in the preboot environment; disabled when an operating system is started
 - Runtime services are always available, even when OS is running
- Note 1: UEFI also includes several earlier standards
 - e.g. ACPI tables are made available to OS loaders, etc.
- Note 2: UEFI specifies the hardware state at preboot
 - e.g. protected mode, flat memory model, all address lines enabled
 - OSes can *expect* hardware to be in a reasonable state at startup

Unified Extensible Firmware Interface (4)

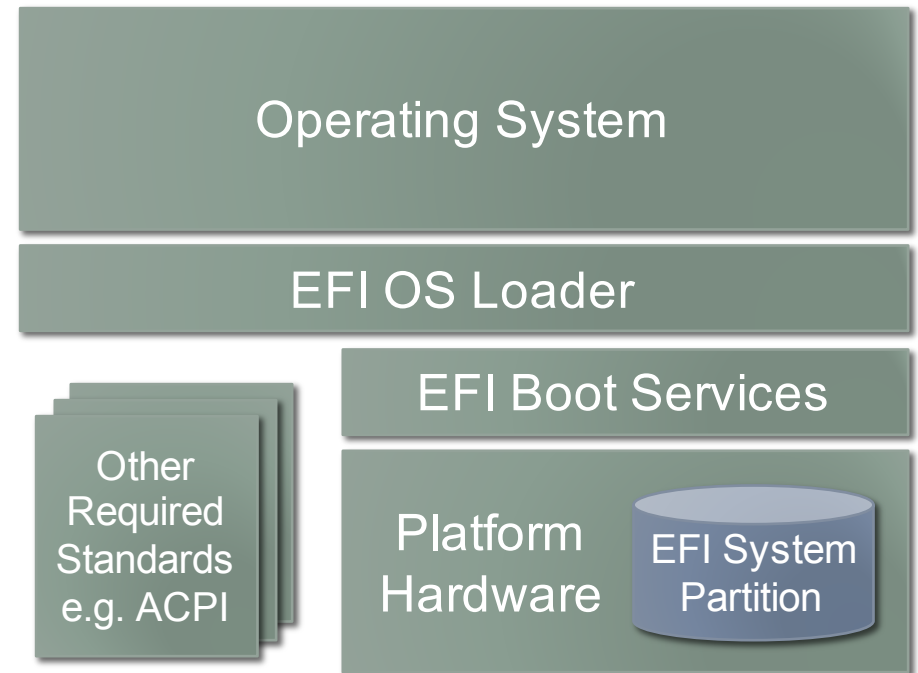
- UEFI also exposes its functionality via tables in memory
- Unlike ACPI and other earlier standards, UEFI includes function-pointers to operations for programs to use
- Example: “Hello World” UEFI application

```
#include <efi.h>
#include <efilib.h>
```

```
EFI_STATUS efi_main(EFI_HANDLE ImageHandle,
                    EFI_SYSTEM_TABLE *SystemTable {
    SIMPLE_TEXT_OUTPUT_INTERFACE *con_out;
    InitializeLib(ImageHandle, SystemTable);
    con_out = SystemTable->ConOut;
    uefi_call_wrapper(conout->OutputString, 2, con_out,
                     (CHAR16 *) L"Hello World\n\r");
    return EFI_SUCCESS;
}
```

Unified Extensible Firmware Interface (5)

- Another picture of UEFI:
- UEFI provides *some* of its functionality in firmware...
- UEFI modules are often stored on a special disk partition



- The EFI System Partition is the first partition of a disk in the system
 - Often a simple format that can be supported in firmware, e.g. FAT32

Hard Disk Addressing

- Another historical bootstrap issue: disk sector addressing
- Cylinder-Head-Sector (CHS) hard disk addressing:
 - BIOS generally allows disks to have up to 1024 cylinders, 255 heads, and 63 sectors/track.
 - These limits grew from earlier, more severe limitations
 - Since sectors were historically 512 bytes: $1024 \times 255 \times 63 \times 512 = \sim 7.8\text{GiB}$ maximum size of a disk that can use CHS addressing
 - “The 8GB limit”
- Logical Block Addressing (LBA, ~1994) was introduced to overcome CHS limits
 - Each sector (“block”) is addressed by an index starting at 0
 - Started out with 22-bit values; most recent ATA-6 spec uses 48 bits
 - Can access disks up to 128PiB

Hard Disk Partitioning

- LBA allows *very* large disks to be used...
- Problem: Master Boot Records specify partitions using 32-bit LBA values (32-bit start, 32-bit size)
 - Again, assuming 512-byte sectors, limits partitions to 2TiB in size!
- Solution: GUID Partition Tables (GPT)
 - GUID = Globally Unique ID, a 128-bit identifier generated to have a high likelihood of being unique
- Partition descriptors use 64-bit LBA values – 9.4 zettabyte (8×10^{21} , or 8 zebibytes = $8 \times 512 \times 2^{64}$ byte) partitions!
- Disks, partition types, partitions all identified by GUIDs
 - Partition-type GUIDs are standardized
 - (MBR uses a 1-byte value to indicate partition type)

GUID Partition Tables

- Also allows up to 128 partitions per hard disk
 - GUID partition table occupies 33 sectors at the start of the disk
- A “legacy MBR” occupies first sector (LBA 0)
 - Includes a single partition that covers the entire disk
 - Partition type is set to a value unused by all major OSes
 - Reason: if a legacy MBR tool is used on the disk, it won't be as likely to mangle the GUID partition table on the disk
- Disks with GUID Partition Tables maintain two GPTs
 - Identical copies, kept at the start and end of the disk
 - Reduces likelihood that corruption will render the disk unusable
- Most modern OSes can use GUID partition tables now
 - Not all of them can boot off of a GPT disk without firmware support

GUID Partition Tables and UEFI

- UEFI specification includes the GUID partition table spec
 - UEFI requires GUID partition tables
- When an OS is installed on a UEFI system, the OS loader is installed into EFI System Partition (using EFI services)
 - Allows the UEFI preboot system to provide multiboot services
- UEFI is still a relatively early-stage standard
 - Not every company follows the UEFI standard precisely
- Example: Apple Intel Macs use UEFI...
 - Firmware basically implements UEFI (with caveats)
 - Disks use GUID partition tables
 - Firmware doesn't use the EFI System Partition to load the OS! Instead, goes straight to a bootable HFS+ partition.
 - EFI System Partition is used for staging/installing firmware updates

For More Information...

- UEFI Standards – <http://www.uefi.org>
- Tianocore – <http://sourceforge.net/mediawiki/tianocore/>
 - Intel's implementation of EFI for several platforms
 - Includes an EFI Development Kit for writing UEFI components
- Windows and Linux both support UEFI platforms
 - GRUB and many other bootloaders understand UEFI
- Can install UEFI bootloaders on Mac OS X (if you dare!)
 - Chameleon – <http://chameleon.osx86.hu>
 - rEFIt – <http://refit.sourceforge.net> (no longer actively maintained)
 - rEFInd – <http://www.rodsbooks.com/refind/> (fork of rEFIt)
- VirtualBox and QEMU can both emulate UEFI hardware

Next Time

- Start exploring the process abstraction